Scientific Programming with High Performance Fortran: A Case Study Using the xHPF Compiler

ERIC DE STURLER[1] AND VOLKER STRUMPEN[2]

[1]*Swiss Center for Scientific Computing, Swiss Federal Institute of Technology (ETH), Zurich, Switzerland; e-mail: sturler@scsc.ethz.ch*
[2]*Department of Electrical and Computer Engineering, University of Iowa, 4400 Engineering Building, Iowa City, IA 52242; e-mail: strumpen@eng.uiowa.edu*

ABSTRACT

Recently, the first commercial High Performance Fortran (HPF) subset compilers have appeared. This article reports on our experiences with the xHPF compiler of Applied Parallel Research, version 1.2, for the Intel Paragon. At this stage, we do not expect very High Performance from our HPF programs, even though performance will eventually be of paramount importance for the acceptance of HPF. Instead, our primary objective is to study how to convert large Fortran 77 (F77) programs to HPF such that the compiler generates reasonably efficient parallel code. We report on a case study that identifies several problems when parallelizing code with HPF; most of these problems affect current HPF compiler technology in general, although some are specific for the xHPF compiler. We discuss our solutions from the perspective of the scientific programmer, and present timing results on the Intel Paragon. The case study comprises three programs of different complexity with respect to parallelization. We use the dense matrix-matrix product to show that the distribution of arrays and the order of nested loops significantly influence the performance of the parallel program. We use Gaussian elimination with partial pivoting to study the parallelization strategy of the compiler. There are various ways to structure this algorithm for a particular data distribution. This example shows how much effort may be demanded from the programmer to support the compiler in generating an efficient parallel implementation. Finally, we use a small application to show that the more complicated structure of a larger program may introduce problems for the parallelization, even though all subroutines of the application are easy to parallelize by themselves. The application consists of a finite volume discretization on a structured grid and a nested iterative solver. Our case study shows that it is possible to obtain reasonably efficient parallel programs with xHPF, although the compiler needs substantial support from the programmer.

1 INTRODUCTION

Our objective is the conversion of Fortran 77 (F77) programs to High Performance Fortran (HPF) programs that allow the compiler to generate efficient parallel code. However, the conclusions of our case study also apply to the development of new HPF programs. We do not know in detail how the FORGE HPF Parallelizer xHPF of Applied Parallel Research (APR) [1] works below the user level, nor whether other compilers will handle certain problems the same way. We try to distinguish between general problems and those that seem to be xHPF specific; in "Flow-Simulation" in Section 3 two problems are discussed that are mainly xHPF specific. However, in general we focus on problems and solutions that are at the core of using HPF, and we believe that the problems

Received October 1995
Revised March 1996

This work was carried out while Volker Strumpen was with the Institute of Scientific Computing, Department of Computer Science, Swiss Federal Institute of Technology (ETH), Zurich, Switzerland.

Scientific Programming, Vol. 6, pp. 127–152 (1997)
CCC 1058-9244/97/010127-26

and solutions that we have found are valuable to other programmers and also to compiler developers.

HPF, which is an extension to Fortran 90 (F90), offers a uniform programming interface for parallel computers that abstracts from the architecture of the machine. It is a data-parallel language where parallelization is addressed by means of explicit array distribution using directives and/or the `forall` statement, implicit loop parallelization, implicitly through array-syntax, as well as intrinsic parallel functions and standard library routines. The programmer specifies the distribution of arrays and the alignment between arrays (the distribution of one array relative to the distribution of another array) by annotating the program with directives. An important aspect of HPF directives is that they are safe; they do not change the semantics of the program, but serve as a hint to the compiler. The programmer specifies (potential) parallelization by using parallel loop statements, F90 array syntax, and several new intrinsic functions and standard library functions. These routines provide both parallel computational functions as well as inquiry functions, which allow explicit adaptation of the program to the run-time configuration. With these parallel constructs, the compiler is expected to generate reasonably efficient parallel code automatically. This especially involves the distribution of arrays, the efficient implementation of communication and synchronization (typically by inserting communication operations in the code), the division of work among the processors, and the use of temporary data structures. One of the most important objectives of HPF is so-called performance portability; an HPF program should display high performance over as large a range of computers and compilers as possible. The HPF language is defined in [2], and a more detailed discussion is given in [3]. A comprehensive introduction is given in [4].

HPF contains many features for which it is still hard to generate efficient programs automatically, because this would require significantly increased quality of interprocedural analysis and optimization. Therefore, to support early implementations of parts of the language and maintain portability, an official subset of the language has been defined as the minimal implementation. We will refer to this subset as the HPF Subset [2].

The xHPF compiler offers the HPF Subset with some exceptions and several additional APR-specific directives. The most important use of the APR directives is to force the compiler to make certain performance optimizations, like forced loop parallelization, or to omit communication or synchronization that would be generated otherwise. In contrast to HPF directives, these directives are dangerous; they may change the semantics of the program, which makes their use undesirable. We refer to these directives collectively as forced optimizations. A more detailed description of the xHPF compiler is given in Section 2.2.

The expectations of the automatic parallelization capabilities of HPF compilers are often high. However, we needed an extensive learning phase until we were able to obtain reasonable performance with still high programming effort. In order to identify the problems that occur when converting an F77 program to an HPF program we did a case study. The case study is based on three programs that reveal several problems that will confront the novice HPF programmer. First, we use a dense matrix-matrix product to show that the distribution of arrays and the order of nested loops significantly influence the performance of the parallel program. This result is important, because loop parallelization and array distribution are the corner stones of HPF. Second, we use Gaussian elimination with partial pivoting to study the parallelization strategy of the compiler. There are various ways to structure this algorithm for a particular data distribution. This example provides some insight into the analysis and optimization capabilities of the compiler, and shows how much effort may be required from the programmer to support the compiler in generating an efficient parallel implementation. Finally, we use a small application to show that the more complicated structure of a larger program may introduce problems for the parallelization, even though all subroutines of the application are easy to parallelize by themselves. The application consists of a finite volume discretization on a structured grid and a nested iterative solver.

Our case study reveals three main results: (1) It is possible to obtain reasonably efficient parallel programs with xHPF. (2) A detailed analysis of the F77 program and sometimes of the HPF compiler-generated program is required, and partial rewriting/restructuring of F77 programs seems necessary. (3) The learning effort is substantial, because the HPF compiler needs significant support from the programmer.

2 MULTIPROCESSOR ARCHITECTURE AND HPF ENVIRONMENT

All experiments presented in this article have been carried out on an Intel Paragon with the APR HPF Subset compiler xHPF [1].

2.1 Intel Paragon

The target architecture for our HPF programs is an Intel Paragon XP/S5+. The Paragon is a distributed

memory multiple-instruction multiple-data (MIMD) multiprocessor, which scales with respect to the number of nodes. Each node consists of two i860XP RISC processors, running at a 50-MHz clock frequency. Each processor accommodates two floating-point pipelines that deliver a peak performance of 75 Mflop/s for double-precision arithmetic. One of the processors functions as a compute processor, and the other as a communication processor. The communication processors are connected via a network with a two-dimensional mesh topology. The main task of the communication processors is the implementation of cut-through routing, which reduces the overhead of routing such that message-transfer times are almost independent of the number of hops.

The Intel Paragon at ETH Zurich [5] consists of 112 nodes, 96 of which are available for running user programs. Each node has a 32-Mbyte memory. The data cache size of the i860 is 16 kbyte. The network has a theoretical bidirectional bandwidth of 200 Mbyte/s. Each node runs a Mach 3.0 micro kernel and the OSF/1 operating system. Intel's NX message-passing library manages communication among user processes. The optimizing `if77` compiler, version 4.5.2, was used to compile all the programs presented in this article, including the xHPF-generated, single-program multiple-data (SPMD)-style programs for parallel execution.

2.2 xHPF

The xHPF compiler offers the HPF Subset with some exceptions and several additional features. The xHPF compiler is a source-to-source compiler, which translates HPF programs (possibly with additional APR-specific directives) into F77 programs with subroutine calls to the APR run-time library. The compiler itself does not offer any support for debugging at the HPF level; debugging is typically done at the F77 level (which is cumbersome). APR does, however, offer additional tools that facilitate debugging.

The two most notable deficiencies of the xHPF compiler with respect to the HPF Subset are that (1) the implementation of the `DISTRIBUTE` and `ALIGN` directives for dummy arguments does not conform to the HPF (Subset) language specification and is much less powerful and (2) the `PROCESSORS` directive and all directives that refer to a `PROCESSORS` declaration are not implemented. If the xHPF compiler encounters a `DISTRIBUTE` or `ALIGN` directive for a dummy argument, it will either make the required distribution for all actual arguments in the routines where they are declared or it will ignore the directive. This leads to problems such as directive-based implicit alignments, which are discussed later, and it obstructs the implementation of the required distribution for arguments that are not an entire data object, e.g., array segments.

Apart from the HPF Subset, the APR xHPF compiler offers several APR-specific directives. The most important use of these APR directives is to force the compiler to make certain optimizations, like forced loop parallelization, or to omit otherwise included communication or synchronization. Although these directives are dangerous—they may change the semantics of the program—the use of these directives can lead to substantial performance improvements. The xHPF compiler offers the following additional directives:

1. `CAPR$ DO PAR` {`ON` *array-element reference*} selects a loop for parallelization. The `ON` clause allows the programmer to let a distributed array (reference) determine the parallelization of the loop iteration over the processors. Using this directive, the programmer can relate the work distribution to the distribution of a data set.
2. `CAPR$ DO NOPAR` indicates that the following loop should not be parallelized; this is important when using the automatic parallelization features of the compiler.
3. `CAPR$ IGNORE ALL INHIBITORS` forces the compiler to parallelize the following loop even if its analysis indicates that this is not possible. This is important since the analysis is often insufficient, and the compiler will not parallelize the loop under the default assumption that it is not safe.
4. `CAPR$ IGNORE` *type* `COM` {`ON` *reference*} indicates that a certain *type* of communication is not necessary for the particular reference, thus improving efficiency.
5. `CAPR$ PARTITION` *array(format)*, `CAPR$ PARTITION` *(format):: array-list*, `CAPR$ PARTITION_NOMOVE` *array(format)*, `CAPR$ PARTITION_NOMOVE` *(format):: array-list:* With these directives the programmer can distribute an array at any place in the program according to a given scheme. They also allow the distribution of a segment of an array. If a "partitioning" is done inside a subroutine, the new distribution will also persist outside the subroutine; so returning to the original distribution must be done explicitly. The `NOMOVE` variant indicates that no communication of data is necessary (e.g., for scratch arrays).

The compiler offers automatic parallelization in essentially two ways. One way is to make a profile with

sequential or preliminary parallel code. The compiler can use this information to determine a parallelization strategy. The other way is to provide some initial array distributions or loop parallelizations (using the APR directives). The compiler will then iteratively try to improve the parallelization by trying to distribute any nondistributed array that is referenced in a parallel loop, and by trying to parallelize any nonparallel loop that references distributed arrays. The programmer can set a maximum number of steps for this iterative improvement strategy. The two strategies can also be mixed.

```
1            double precision a(m,n), b(n,m), c(m,m)
2      *
3      chpf$ distribute a (block,*)
4      chpf$ distribute b (block,*)
5      chpf$ distribute c (block,*)
6      *
7            do 40 i = 1, m
8               do 40 j = 1, m
9                  do 40 k = 1, n
10                    c(i,j) = c(i,j) + a(i,k) * b(k,j)
11         40 continue
```

FIGURE 1 Program fragment of matrix-matrix multiplication.

3 CASE STUDY

We analyze the HPF Subset as implemented by APR using three programs. (1) A dense matrix-matrix product, which shows that the distribution of arrays and the order of nested loops in which the distributed arrays are referenced determine the performance of the parallel program. Specifically, the loop order affects the place of subroutine calls for communication and run-time checks generated by the HPF compiler, which determines the cost of communication. Furthermore, these subroutine calls influence the opportunities of the native compiler for optimization. (2) Gaussian elimination with partial pivoting and backward substitution. We examine the parallelization strategies of the compiler and describe data-parallel implementations that support the compiler in generating an efficient parallel implementation. (3) A small application, which shows that the more complicated structure of a larger program may impede efficient parallelization, even though all subroutines of the application are easy to parallelize individually.

This section is structured as follows: Section 3.1 introduces the programs and their implementation using F77. Section 3.2 presents the HPF versions of the programs, the background for the conversion from F77 programs, and the development of the HPF programs. An experimental analysis of the three programs is presented in Section 3.3. Section 3.4 summarizes the results and observations obtained from each specific parallelization.

3.1 F77 Programs

Dense Matrix-Matrix Multiplication

Dense matrix-matrix multiplication is one of the basic building blocks of linear algebra. F90 offers an intrinsic matrix-matrix multiplication function (MATMUL). This function is also in the HPF Subset, but it is not included in the xHPF version discussed here. However, our interest is not in the matrix-matrix multiplication by itself, but in the effects of loop ordering, communication, and processor utilization. Although the sequential algorithm comprises a straightforward threefold loop, optimizations for better sequential performance as well as for parallelization are far from trivial. Structuring optimized code manually for RISC machines with virtual memory hierarchies does not only complicate the code, but may also prevent the compiler from recognizing potential optimizations apparent in the simple code. We experienced this anomaly when we compared the simple matrix-matrix multiplication with the `dgemm` level 3 BLAS routine, a well-known optimized code for dense matrix-matrix multiplication [6]. The optimizing `if77` compiler generates code that is approximately three times faster with the straightforward threefold loop than with the manually optimized `dgemm` routine. All sequential run-times reported in Section 3.3, subsection "Dense Matrix-Matrix Multiplication," have been measured with the straightforward threefold loop version. The *ijk*-version is shown in Figure 1.

Gaussian Elimination

Our second example is the solution of a dense system of linear equations by Gaussian elimination with partial pivoting and backward substitution. We start our investigation with a *kij*-forward-looking Gaussian elimination and a *ji*-version of the backward substitution [7]. The sequential code is shown in the Appendix, instrumented with HPF directives. In order to include the impact of partial pivoting on the performance, we use an artificial, square system of order 1,000 that yields the worst case of 999 row exchanges.

The partial pivoting complicates the elimination code so that a complete empirical analysis of all possible parallel implementations is not sensible. We, therefore, concentrate on the implementation decisions of

the parallelizing compiler and the effort the programmer has to invest in order to support the generation of efficient parallel code. For the matrix-matrix product, the implementation follows immediately from the data distribution and the loop order, even if this does not necessarily produce the desired efficiency. In this respect the parallelization of the dense matrix-matrix product is straightforward compared to the parallelization of the Gaussian elimination with partial pivoting, where several choices need to be made beyond the specification of the data distribution. These choices affect the communication cost of the program significantly. This example illustrates the process of performance tuning that eventually leads to a *good* choice.

Flow-Simulation

The third example involves finite volume discretization of a simple flow problem on a regular grid and a fixed number of steps of the nested iterative (linear) solver described in [8].

The finite volume discretization computes the set of linear equations that needs to be solved. This involves integration over a volume in the grid or its boundary using data that is defined over this volume and neighboring volumes. The algorithm is embarrassingly parallel. Since we use a regular grid, load balancing is straightforward.

The iterative solver involves matrix-vector products, inner products, and vector updates. For simplicity we did not use any preconditioning at this stage. Of course, for realistic simulations, preconditioning is of great importance. However, for reasonable preconditioners (with respect to both mathematical properties and parallel properties), we need an inquiry function for the actual distribution of the matrix. Such functions were not yet available in the version of xHPF that we used: the current version has a limited, nonconforming, distribution inquiry function. The matrix is stored in diagonal format, which allows straightforward parallelization. The vector updates, using the daxpy routine, are in principle embarrassingly parallel, but some unexpected problems are described in Section 3.2, subsection "Flow-Simulation." The inner products are also simple to parallelize; however, the global communication necessary for the reduction add may have a negative influence on the scalability of the solver; especially because this particular solver relies heavily on orthogonalization [9]. In principle, this problem is inherent to the iterative solver and not to HPF. However, the techniques that can be exploited in message-passing programming to reduce the loss of efficiency, like latency hiding, cannot be used explicitly in HPF, nor does it seem that such techniques are currently used automatically in HPF compilers.

3.2 Parallel Programming with xHPF

We now discuss the parallelization issues of our programs. Due to the varying degree of complexity, the number of topics discussed with each program varies.

Dense Matrix-Matrix Multiplication

Parallelization of the dense matrix-matrix multiplication is implemented by block partitioning the matrices. We distribute all three matrices in order to ensure scalability with respect to memory requirements. More specifically, we choose a row-block distribution for all matrices. Figure 1 shows the ijk-variant of the threefold loop that implements the multiplication, including the HPF data distribution directives in lines 3–5.

Given the data distribution, the communication pattern of this program depends on the loop order. Since parallelization of loops is a basic technique employed in parallelizing compilers, we expect compiler optimizations to minimize the communication cost. Ideally, the run-time should be independent of the loop order. The results presented in Section 3.3, subsection "Dense Matrix-Matrix Multiplication," show that this is not the case.

Gaussian Elimination

We first consider the parallelization of the Gaussian elimination and backward substitution algorithms without any algorithmic changes to assist an HPF compiler (see the Appendix). We choose a column cyclic distribution for the matrix, and replicate the right-hand side on all processors: the HPF distribution directives at lines 18 and 19 request this. Running the solver with these HPF directives only (without the `capr$` directives) produces the execution times given in Table 1 (distribution only), which are clearly unsatisfactory. The execution time increases proportionally to the number of processors. This poor parallelization results from the fact that the compiler inserts communication and run-time checks inside the loops, thereby generating tremendous overhead.

We first try to improve the performance using APR-specific forced optimizations (`ignore` directives) to move communication out of the loop (line 30), to distribute a loop such that its body is executed on the processor that owns the required elements (lines 29, 54, 70, and 75), and to prevent superfluous communication and run-time checks (lines 71 and 76). We regard this approach as the APR-specific way to resolve parallelization problems. The execution times of this version, given in Table 1 (forced optimization), are encouraging. However, this version still contains unnecessary communication. The reason is that the

Table 1. Execution Times (Seconds) of Gaussian Elimination and Backward Substitution for a Nonsymmetric 1000 × 1000 matrix and 999 Row Exchanges due to Partial Pivoting

No. of Procs	Distribution Only		Forced Optimization	
	Elim	Back	Elim	Back
1(seq)*	55.88	0.01	55.88	0.01
1	769.14	1.14	137.83	0.86
2	1599.49	2.06	95.84	2.40
4	2329.41	2.23	41.46	2.26
8	3955.12	3.05	32.90	3.06
16	7595.05	4.57	25.40	4.59
25	—	—	24.76	6.10
32	—	—	25.30	7.74
50	23096.44	10.51	28.76	10.79
64	—	—	32.56	13.43
80	—	—	37.15	16.22
96	—	—	41.83	19.04

Note. The run-times show a large difference between the parallelization using data-distribution directives only (distribution only), and with additional forced optimizations (forced optimization). For the distribution only version we did not measure the execution times for all numbers of processors.

* cf. Table 4.

scope of automatic parallelization and optimization is limited to single loops; optimizations over several consecutive loops are not considered. Also, the reuse of data that have been fetched previously is ignored. In order to indicate better implementations and optimizations to the compiler, we have to provide it with programs that fit an HPF-like data-parallel programming style (see e.g., [10], [11]).

We will now discuss the general design decisions for a data-parallel implementation. Since we distribute the columns of the matrix cyclically, the row exchange for pivoting can be performed in parallel without communication. Furthermore, after the elimination factors have been computed and are available on all processors, the distributed update of the submatrix rows and the replicated right-hand side with the pivot row is also local. Therefore, communication is necessary only to provide the index of the pivot row and the elimination factors to all processors. Hence, the pivot column (the pivot element and the elements below) and/or the elimination factors must be broadcast to the other processors from the processor that owns the pivot column.* This leads to the following three parallelization variants for one iteration of the outermost loop (loop 200 in the Appendix):

1. (See program in the Appendix). The processor that owns the pivot column does the pivot search (lines 27–36) locally and broadcasts the result. Next, the row exchange (lines 38–66) is carried out in parallel. Then, the processor that owns the pivot column computes the elimination factors (lines 72–74) and broadcasts the column which contains these. Afterward, the update of the submatrix (lines 77–82) is carried out in parallel, and the update of the right-hand side (line 81) is replicated on all processors.
2. (See program in the Appendix). First, the pivot column is broadcast (lines 33–35) and the pivot search (lines 39–46) is replicated. Then the row exchange (lines 58–79) is carried out in parallel. The computation of the elimination factors (lines 83–85) is also replicated on all processors, and the update of the submatrix (lines 86–93) is carried out in parallel. The update of the right-hand side (line 92) is replicated on all processors.
3. (This scheme has not been implemented). The processor that owns the pivot column does the pivot search locally and broadcasts the result. Next, the row exchange is carried out in parallel. Then, the pivot column is broadcast, and the computation of the elimination factors is replicated on all processors. Afterward, the update of the submatrix is carried out in parallel, and the update of the right-hand side is replicated on all processors.

The three variants can be implemented with approximately the same efficiency, because they exhibit similar communication. The second variant should be the most efficient, because it needs slightly less communication than the others and it requires the least synchronization. We consider two possible changes to the source program that establish a data-parallel programming style:

1. Local pivot search: We replace the scalars `ppiv` and `abspiv` by arrays of a size that equals the number of columns, and align these arrays with the columns of the matrix. This way, for each column `col` of the matrix, we use the local vector elements `ppiv(col)` and `abspiv(col)` in the pivot search instead of (replicated) scalars

* It is also possible to distribute the pivot column and the computation of the elimination factors. We will not consider this possibility because it will be very difficult for the compiler to analyze whether this improves efficiency or not.

that incur communication inside the loop.* Now, only the final result must be communicated to the other processors. This implements the first scheme.

2. All local: We declare a replicated (dummy or automatic) array (`pcol`) of the size of a matrix column, and copy the pivot column into that array; see Appendix, loop 400. This copying leads to the broadcast of the pivot column. Furthermore, we replace all references to the pivot column by corresponding references to the replicated array (`pcol`). This leads to strictly local work for the pivot search, the row exchange, the computation of the elimination factors, and the update of the rows and the right-hand side. This implements the second scheme.

The experimental results obtained with these data-parallel implementations are presented in Section 3.3, subsection "Gaussian Elimination."

Flow-Simulation

In a large application, a single array may be passed to and updated by many different subroutines with different access patterns. Conversely, a single subroutine may be called at many places in the program with array arguments that differ in size, shape, and/or distribution and alignment. This leads to a complex set of requirements on the implementation of subroutines (especially the loops over distributed array-arguments) and the distribution and alignment of their dummy arguments if an efficient code is to be produced. None of the specific parts of the program plays a role by itself; indeed, each of the routines used in this program parallelizes well by itself. It is the global parallelization that causes the problems.

First, we describe some concepts that underlie the parallelization of loops and the distribution of arrays. For simplicity, we will not include replication in this description; for extensions see [12]. Different compilers will address these concepts in different ways. However, since the distribution of arrays and parallelization of loops lie at the heart of data parallelism, the concepts themselves must be addressed in some way. Following a general introduction, we describe the problems we found for the APR compiler. Two of these problems seem to be xHPF specific, the directive-based implicit alignment and the interprocedural propagation.

With each array we can associate an index set that consists of the valid index references to that array. With each loop we can associate an index set that consists of the values that are assigned to the loop index during the execution of the loop. The distribution of an array `a` is described by a function from the array index set into a processor index set P, $M_{I_a,P}$: $I_a \rightarrow P$.† Likewise, we can describe the distribution of a loop `l` by a function from the loop index set to a processor index set P, $M_{I_l,P}$: $I_l \rightarrow P$. We assume that all statements within one iteration of the loop are carried out on the same processor. This is the case for the APR xHPF compiler and for the Oxygen compiler [13]; however, it is not defined in the HPF language specification [2]. Through this use of index sets, the distribution of data and the parallelization of loops are described in the same way.

The programmer can indicate to the compiler that the distribution of the elements of one array should depend on the distribution of the elements of another array through the alignment directive [2]. This directive describes a linear function from the index set of one array, the **alignee,** into the index set of another array, the **align target.** Given arrays `a` and `b` with the index sets I_a and I_b, the alignment α: $I_a \rightarrow I_b$, a processor set P, and the "distribution" $M_{I_b,P}$ of the array `b`, we can describe the interpretation of the alignment indicated by α. For the processor set P, this alignment leads to the distribution

$$M_{I_a,P}: I_a \rightarrow P \text{ such that } M_{I_a,P}(i) = M_{I_b,P}(\alpha(i)).$$

In order to minimize data movement for the execution of a distributed loop, an optimizing compiler will try to assign the execution of each loop iteration to the processor that owns the array elements referenced in that loop iteration. This implies functions from the loop index set to the index sets of the arrays that are referenced inside the loop, which leads to certain desired "alignments" between the loop and the array index sets. In fact, for the APR xHPF compiler, the alignment of the loop index set with one array index set can be indicated explicitly through an APR-specific directive.

In this article we will refer to alignments that are indicated explicitly by directives as explicit alignments. We will refer to all other alignments that an optimizing compiler may make to reduce communication as implicit alignments. Such implicit alignments

* According to the data-parallel programming paradigm, scalars are replicated on all processors and therefore are broadcast if they are updated on some processor.

† We use the term *processor* for simplicity; one may use instead process, thread, and so on. These have to be mapped to physical processors in turn.

come, for example, from references to different distributed arrays in a single distributed loop. Several loop parallelization strategies are used in data-parallel compilers that lead to different implicit alignments, e.g., *owner computes* and *almost owner computes* [14]. However, they all suffer from serious limitations with respect to optimization when applied rigorously, so in practice relaxed versions are used. APR uses a so-called *owner sets* strategy [1].

We illustrate the effects of loop parallelization by means of an example. Consider a loop in which two arrays are referenced once. The loop index set is given by I_l, the index sets of the arrays are I_a and I_b, and the alignments of the loop with the array index sets are given by the functions λ_a: $I_l \rightarrow I_a$ and λ_b: $I_l \rightarrow I_b$. In order to keep all array references local in this loop, the distributions $M_{I_a,P}$: $I_a \rightarrow P$ and $M_{I_b,P}$: $I_b \rightarrow P$ must fulfil, for all possible P,

$$\forall i \in I_l: M_{I_a,P}(\lambda_a(i)) = M_{I_b,P}(\lambda_b(i)).$$

This defines an implicit alignment of I_a and I_b in the following sense,

$$\exists \alpha_b: \lambda_a(I_l) \rightarrow I_b \text{ such that } \alpha_b(\lambda_a(i)) = \lambda_b(i)$$

or

$$\exists \beta_a: \lambda_b(I_l) \rightarrow I_a \text{ such that } \beta_a(\lambda_b(i)) = \lambda_a(i).$$

Obviously implicit alignments can have a much more complicated form than explicit alignments, and they may be only a relation on part of the index set of the alignee. Implicit alignments may also arise indirectly by aligning several arrays explicitly with the same array, or through an explicit alignment in combination with a distributed loop referencing more than one distributed array. Although implicit alignments indicate potential compiler optimizations, they can create problems as well, as we discuss below.

We now describe four problem classes that occurred in the parallelization of our application.

1. Directive-based implicit alignment. Subroutines may declare the alignment and distribution attributes of dummy arguments. These declarations can lead to implicit alignments of actual arguments of the subroutine in separate calls. This creates problems for the xHPF compiler in particular because of its poor implementation of the `DISTRIBUTE` and `ALIGN` directives for dummy arguments. Since the xHPF compiler must either move the distribution of actual arguments to where they are declared or ignore the directive, calling such a subroutine with different actual arguments will only lead to the desired distributions and alignments if all actual arguments are or can be aligned.

   For other compilers this would at most constitute a potential optimization problem. Most compilers will just distribute the associated dummy argument on entry to the subroutine according to the given directive and redistribute upon exit from the subroutine. Possible inefficiencies are then entirely the programmer's responsibility, and do not form a problem for the compiler.

   Consider a scenario where subroutine `sub(x,y)` aligns array `x` with array `y` on subroutine entry by means of an `align` directive. Then, two calls to this subroutine of the form `call sub(a,b)` and `call sub(c,b)` lead to an implicit alignment of the arrays `a` and `c`. The compiler may want to propagate the two alignments to the calling routine to prevent redistribution inside the subroutine (as the APR xHPF compiler must do). This will only be possible if the potential distributions (as indicated by the programmer, for example) of the index sets of `a` and `c` satisfy the requirements from this implicit alignment.

   We call the implicit alignment of arrays that arises in this way directive-based implicit alignment, because it arises out of alignments that are indicated by a directive.
2. Loop-based implicit alignment. As explained in the case of a distributed loop containing references to a distributed array, an optimizing compiler will try to align the loop index set with the index set of the distributed array so as to minimize communication. We refer to this type of implicit alignment as loop-based implicit alignment. If such a loop occurs inside a subroutine and the distributed array is a dummy argument, this leads to the implicit alignment of the loop index set with the index set of the actual argument in each call to this subroutine. If the distribution of the loop index set is different for each call to the subroutine, this would not create any problems. However, this seems generally not to be the case. Except for the specific processor set, which is only known run-time (at loading), the distribution mapping will be fixed by the compiler, because it typically generates only one object module for the subroutine. This holds, for example, for the APR xHPF compiler, for the PGI compiler [L. Meadows, Personal Communication] (we tested this ourselves), and for the ADAPTOR-tool [15]. A fixed distribution

mapping of the loop index set, however, leads to the implicit alignment of all the actual arguments in separate calls to the subroutine. The effect is the same if all the actual arguments were referenced in this loop at the same time.

3. Aliasing. If more than one symbolic name to a reference (variable or memory location) exists within a loop, we call this an alias, as in the APR xHPF documentation [1]. Aliasing creates problems for loop parallelization if the name is assigned a value within the loop. In practice, the compiler may also spot many "potential" aliases that are not real. This depends largely on the (interprocedural) analysis capabilities of the compiler, and is therefore compiler dependent. The xHPF compiler seems rather sensitive in this respect. Fake aliases lead to parallelization and optimization problems. One potential way to resolve these problems is the use of segments (array syntax), so that the compiler can check that different parts of an array that are passed to a subroutine do not overlap. If this does not work, such problems can be resolved (with the xHPF compiler) with forced optimizations.
4. Interprocedural propagation. Because of the interprocedural analysis of the xHPF compiler and the way this compiler handles the distribution and alignment of dummy arguments, parallelization problems may propagate through the entire program. A problem that comes up in one subroutine may prevent an array from being distributed, which leads to problems in other subroutines, and so on. The interprocedural analysis may have the unwanted side effect of globalizing local problems. This seems mainly an xHPF-specific problem, because most other compilers do not offer interprocedural analysis at this point.

In the following, we illustrate the four problem classes by examples that arose from the parallelization of our application program.

Directive-Based Implicit Alignment. First, we consider potential conflicts arising from directive-based implicit alignment, and then, as a special case, we consider a distribution problem associated with workspace arrays. Consider a simplified version of the subroutine daxpy* (vector update), which is invoked twice within another routine. See the program fragment given in Figure 2.

```
 1        subroutine daxpy(n,da,dx,dy)
 2  c
 3  c     compute dy = da*dx + dy
 4  c     da is a scalar, dx and dy are vectors
 5  c
 6        double precision dx(n), dy(n)
 7        double precision da
 8  chpf$ inherit dy
 9  chpf$ align dx(k) with dy(k)
10  c
11        do i = 1, n
12           dy(i) = dy(i) + da*dx(i)
13        end do
14        end
15  c
16        program use_daxpy
17        double precision a(n), b(n), c(n)
18        double precision alpha, beta
19  chpf$ distribute a(block)
20  chpf$ distribute c(cyclic)

                ⋮

21        call daxpy(n,alpha,b,a)
22        call daxpy(n,beta,b,c)

                ⋮

23        end
```

FIGURE 2 The daxpy routine and its use lead to directive-based implicit alignment.

The explicit alignment of dummy arguments in the daxpy subroutine and the two calls in the program lead to an implicit alignment of the arrays a and c. The daxpy can be executed without any communication if the arguments have the same distribution; the actual distribution of the arguments does not matter. However, in the program in Figure 2, because of the "awkward" implementation of the ALIGN and DISTRIBUTE directives by the xHPF compiler, the implicit alignment leads to a problem regarding the distribution of the array b, and it depends on the rest of the program what the best choice will be for the distribution of b. In fact, for the xHPF compiler the three arrays have to be aligned outside the subroutine for efficient execution. A conflict like this does not only arise with different distributions, it also arises for arrays with the same distribution but with a different

* The daxpy routine as defined in the BLAS has different increments for the two vectors dx and dy; this makes alignment impossible in the general case.

```
1        subroutine sub1(n,vector,dummy_vector)
2  c     dummy_vector is a workspace array
3  chpf$ align dummy_vector with vector
4  c
5        forall (i=1:n) (dummy_vector(i)=vector(i))
           ⋮
6        end
7  c
8        program
9        double precision a(n), b(n), h(n)
10 chpf$ distribute a(block)
11 chpf$ distribute b(cyclic)
           ⋮
12       call sub1(n,a,h)
13       call sub1(n,b,h)
           ⋮
14       end
```

FIGURE 3 Example program illustrating unnecessary implicit alignment.

size or shape, and so on. In the small application under consideration we had to resolve such problems several times.

A problem that will arise frequently in converting existing F77 programs to HPF, and which can easily be avoided, is the implicit alignment of unrelated arrays due to the use of workspace arrays passed to a subroutine. Consider the program fragment in Figure 3. The programmer wants array a to be block distributed (line 10) and array b to be cyclically distributed (line 11). If the parallelizing compiler obeys these directives, any distribution for array h will lead to large communication overhead. Typically, the parallel program will redistribute h upon entry to and exit from the subprogram.

The alignment directive does not solve the problem; it only indicates to the compiler that it is probably better to align (redistribute) the array dummy_vector at the start of the subroutine than to do the communication in the subroutine element-wise. The solution is to use the dynamic allocation features of HPF (derived from F90), i.e., to use automatic arrays as workspace.

Loop-Based Implicit Alignment. Second, we look at a problem associated with loop-based implicit alignment. In the daxpy routine given in Figure 2 we replace the prescriptive (desired) alignment [2] at line 9 by a descriptive (asserted) alignment [2]. The descriptive alignment (chpf$ align dx with *dy) asserts to the compiler that the actual arguments are (already) aligned on entry to the subroutine. Now, consider the use of the daxpy in the program fragment of Figure 4.

For each call to daxpy, the compiler knows that the actual arguments are aligned and that no communication is necessary. However, the daxpy loop from the first call (line 7) should be distributed due to the block-distribution directive in line 5, whereas the daxpy loop from the second call (line 8) should be replicated due to the replicated-distribution directive in line 6.

The APR xHPF compiler generates a single implementation of the daxpy routine (and so does the PGI compiler [L. Meadow, Personal Communication] [we tested this ourselves] and the ADAPTOR tool [T. Brandes, Personal Communication]). This results in a compiler decision either to replicate the loop, which gives (large) communication and calculation overhead for the distributed arrays, or to parallelize the loop, which leads to large communication overhead for the replicated arrays. Such an implementation treats unrelated arrays that are actual arguments in different calls of the subroutine as if they were in the same loop; this causes the implicit alignment of a, b, c, and d. We could solve these problems in three ways: Replace the call to daxpy by an equivalent F90 or HPF statement; use two different daxpy routines, one for block-distributed array arguments and one for replicated array arguments; and use xHPF-specific forced optimizations. Each of these options has its own drawbacks

```
1        program loop_align
2        double precision a(n), b(n)
3        double precision c(m), d(m)
4  c
5  chpf$ distribute (block) :: a, b
6  chpf$ distribute (*)     :: c, d
           ⋮
7        call daxpy(n,alpha,a,b)
8        call daxpy(m,beta ,c,d)
           ⋮
9        end
```

FIGURE 4 The daxpy routine and its use lead to undesired loop-based implicit alignment.

(see below), but with current compilers (at least the three mentioned above) one has to compromise to obtain reasonable performance.

It is interesting to note that if the daxpy program were not in a subroutine but inlined by the programmer, there would be no problem at all. The compiler could have implemented the best choice for each loop. For the implementation of a simple routine like the daxpy, this may be a good idea; the entire (simplified) daxpy can be implemented in a single forall- or array-syntax statement. The same problem also appears for more complicated kernel routines that are not suitable for inlining at the source code level.

It is not defined in the HPF standard that a subroutine must have only one object module for all invocations. However, given that the three mentioned compilers do this, it seems to be the "standard" implementation method. Having different subroutine sources for different argument distributions, alignments, or shapes is undesirable because it means that the programmer has to keep track of the different uses, and has to determine the different uses for an existing program. This is difficult or even impossible since the distributions of arrays may not be known before the program is compiled (the compiler may choose distributions other than indicated in the program) or even before run-time.

Aliasing. Now, we consider parallelization and optimization problems that arise from aliases. In F77, programmers often address parts of arrays as independent items. For example, we may consider a two-dimensional array as a matrix and access its "columns," or we may consider a three-dimensional array as a three-dimensional grid and access planes or lines (e.g., in a three-dimensional FFT). If we pass two (or more) different parts of a distributed array to a subroutine and modify at least one part in a parallelized loop, potential aliases occur and the compiler has to determine whether they are real or not. If the compiler cannot establish that no alias exists, it will typically execute the loop sequentially. It may be possible to solve this problem by passing segments of arrays to the subroutine, so that the compiler can check that no actual alias occurs.

In our application, alias problems arise in the generation of orthogonal bases in the iterative solver. A two-dimensional array is used to represent a sequence of vectors; these vectors are generated by multiplying the last generated vector with a matrix and then orthogonalizing the result on the previously generated vectors (see Fig. 5). In the subroutines for the matrix-vector product and the vector update, two (disjunct) parts of the same array are used and one is updated. If the

```
1       subroutine gmreso(n,vv,...)
2  c
3       double precision r(n), vv(n,m+1)
4       double precision hh(m+1,m)
5  c
6  chpf$ distribute r(block)
7  chpf$ align vv(i,*) with r(i)
8  chpf$ distribute (*,*) hh
9  c
                 :
10      ***** Arnoldi *****
11      do i = 1, n
12         vv(i,1) = r(i)/res_norm
13      end do
14 c
15      do i = 1, m
16 c       ***** v_{i+1} = Av_i *****
17         call matvec(n,vv(1,i),vv(1,i+1),...)
                 :
18         do j = 1, i
19 c          ***** h_{ij} = < v_j , v_{i+1} > *****
20            hh(j,i) = ddot(n,vv(1,j),1,vv(1,i+1),1)
21 c          ***** v_{i+1} = v_{i+1} - h_{ij}*v_j *****
22            call daxpy(n,-hh(j,i),vv(1,j),1,vv(1,i+1),1)
23         end do
                 :
24      end do
                 :
25      end
```

FIGURE 5 Parallelization problems due to aliases created by passing parts of vv as different vectors.

compiler cannot check that these parts are disjunct, it cannot parallelize the loops in these subroutines. Using dynamic allocation, we constructed an implementation of the program that enables the compiler to check that the referenced parts of the array are always disjunct; however, the compiler still refused to parallelize the routines. We could solve the problem only with forced optimizations.

An alternative implementation in (full) HPF may be to represent the sequence of vectors by an array of structures, each of which contains a pointer to a vector. However, pointers are not part of the HPF Subset and the construction is rather cumbersome. Moreover, this approach does not provide a solution if it is necessary to traverse a higher dimensional array in more than one way. Consider, for example, a three-dimensional FFT over a three-dimensional array. We would like to consider such an array first as a collection of x-vectors, then y-vectors, and finally z-vectors. This cannot be done with pointers in HPF.

Table 2. Execution Times (Seconds) of Sequential Matrix-Matrix Multiplication (500 × 500): The Sequential Program (Seq) and a One-Processor xHPF-Parallelized Version (Par-1) are Compiled with Two Optimization Degrees of Intel's Native `if77` Compiler

	`-O`		`-full*`	
Loop Order	Seq	Par-1	Seq	Par-1
ijk	159.91	160.22	10.45	138.14
ikj	272.31	272.57	10.45	276.88
jik	157.26	158.67	10.45	135.66
jki	42.19	43.44	10.45	31.96
kij	272.39	272.35	10.45	275.69
kji	44.50	45.08	10.45	35.12

* `-full` is an abbreviation of `-O4 -Mvect -Mstreamall`.

Interprocedural Propagation. This section considers the influence of parallelization problems in one subroutine on the rest of the program. Because of the interprocedural analysis of the xHPF compiler and the way this compiler handles the distribution and alignment of dummy arguments, parallelization problems may propagate through the entire program. Consider the program fragment in the Appendix (see "Propagation"). Potential aliases for the different parts (vectors) of the array `vv` in the Arnoldi part in the subroutine `gmreso()` prevent the compiler from parallelizing the matrix-vector product and the `daxpy`. This, in turn, prevents the distribution of arrays `uu` and `cc` and creates huge communication overhead. The resulting replication of the arrays `uu` and `cc` gives rise to problems in parallelizing other loops, and so on. Finally, significant parts of the code are not parallelized, or suffer from excessive overhead and communication. The propagation has the additional problem for the programmer that it is not always clear where the problem starts, and how it can be solved or at least limited to a small part of the program. The poor handling of the distribution and alignment of subroutine arguments by the xHPF compiler together with the interprocedural analysis has the unwanted effect of making local problems global.

3.3 Experimental Analysis

Dense Matrix-Matrix Multiplication

All timings listed in this section give the run-time of the multiplication kernel, i.e., loop 40 in Figure 1 or one of its permutations. These permutations are defined by changing the order of lines 7–9. Tables 2 and 3 show measurements with all six permutations of the loop order. Table 2 with one processor and Table 3 with 32 processors. We use three square matrices of dimension 500.

Both the sequential F77 program and the xHPF-generated parallel program have been compiled with the native `if77` compiler using two different optimization levels. Option `-O` optimizes register allocation and performs global optimizations such as induction recognition and loop invariant motion. `-full` is an abbreviation for the three options: Optimization level `-O4` extends `-O` by introducing software pipelining and simultaneous use of the integer and the floating-point unit. `-Mvect` enables vector optimizations, and `-Mstreamall` streams all vectors to and from cache in a vector loop. Such optimizations cannot be made manually in the F77 source program.

Comparing the sequential run-times with little (`-O`) and full compiler optimization (`-full`) in Table 2, we make two observations. First, for the original sequential code (`seq`) of the F77 program, the loop order has a large influence on the performance, unless the native compiler (`if77`) is allowed to restructure the loop order. The compiler only restructures the loop order when full compiler optimization is enabled. This improves the computational speed by more than a factor of 4—with the *ikj*- and *kij*-variants up to a factor of 26—and removes the performance dependence on the loop order. Second, for the parallel code executed on one processor (`par-1`), we see that even with full optimization, the native compiler is no longer able to restructure the loop order, because of the code (subroutine calls) inserted by the xHPF compiler (run-time checks, calls to communicate routines, etc.). Therefore, the loop order affects the run-time efficiency of the parallel program. Comparing the sequential and the one-processor run-times of the xHPF-generated parallel code, both fully optimized, we can interpret the difference as parallelization overhead. This overhead is significant.

Table 3. Execution Times (Seconds) of Different Loop Permutations of the Parallel Matrix-Matrix Multiplications (500 × 500) on 32 Procesors

Loop Order	`-O`	`-full`
Cijk	3.32	2.61
Cikj	6.65	6.75
jCik	9.03	8.14
jRki	21.02	20.78
Rkij	13.20	13.17
Rkji	8.20	7.90

Note: All matrices are row-block distributed. `-full` is an abbreviation of `-O4 -Mvect -Mstreamall`.

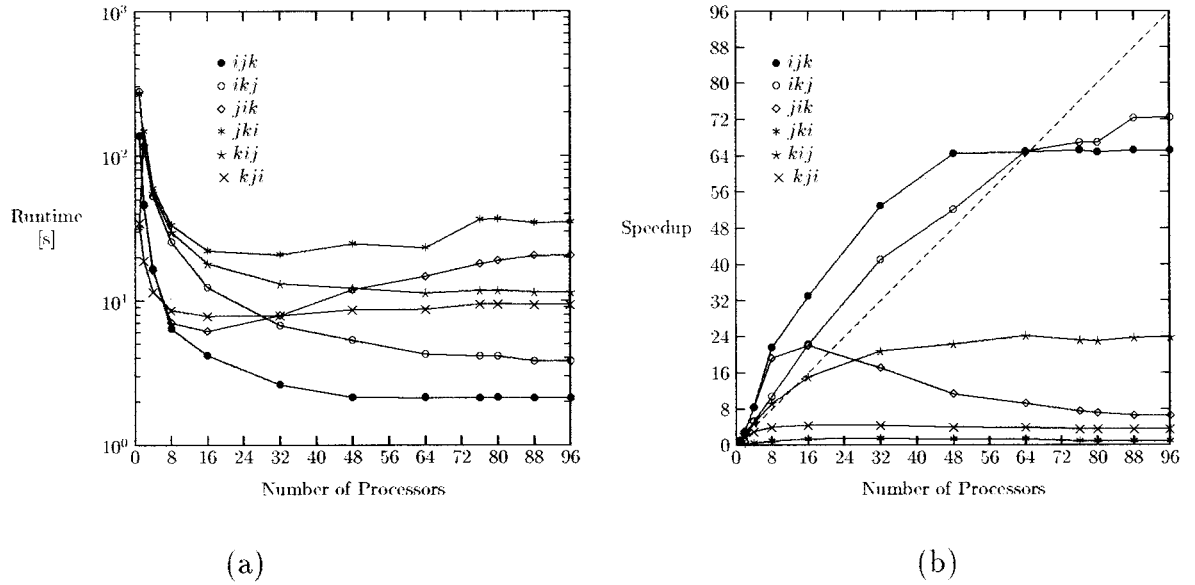

FIGURE 6 Run-time and speedup of dense matrix-matrix multiplication (500 × 500).

Table 3 shows execution times of the parallel code executed on 32 processors. The *C* and *R* extensions of the loop order triplets represent type and location of interprocessor communication inserted by xHPF. *C* denotes preloop communication and *R* denotes a reduction add operation. This information originates from the xHPF-generated F77 program, which is instrumented with calls to the APR run-time library. The run-times with the two different optimization levels hardly differ, indicating that communication overhead is the reason for the variation of execution time. Surprisingly, the best loop order for the parallel code on one processor (*jki*) in Table 2 gives the worst run-time on 32 processors in Table 3, whereas the best code on 32 processors (*ijk*) performs poorly on one processor. To investigate this effect, we measured a series of speedup curves for all permutations of the loop nesting. The results with fully optimized native compilation are presented in Figure 6. These results indicate that in general no single program will be best or even good for all possible numbers of processors.

The superlinear speedups are caused by cache effects, not by swapping. Since the size of all three matrices is only 500 × 500, each processor could hold the three matrices simultaneously; each matrix requires 2 Mbyte, whereas each Paragon processor has 32 Mbyte of main memory [5]. The *jki*-variant also behaves irregularly in another way: The run-time of the two-processor execution (154 s) is about five times larger than that of the one-processor execution of the parallel code (cf. Table 2). We do not know why this happens only with the *jki*-variant. The lack of knowledge of functions in the parallelized code that are supplied with the APR run-time library makes the interpretation of such effects difficult or even impossible.

An additional concern is that for the *ijk* and *ikj* variants, all communication is performed in one operation outside the outermost loop (cf. Table 3). This indicates that at least one of the matrices is replicated on all processors, and the compiler potentially reintroduces the scalability problem with respect to memory requirements that we tried to avoid by distributing all matrices. It is not clear whether this code is scalable with respect to the matrix dimensions.

Gaussian Elimination

To demonstrate the influence of the loop parallelization match and the optimization gap, we measure the execution times of two sequential variants: a *kij*-ordered, forward-looking Gaussian elimination (gelim) combined with an *ij*-backward substitution (backsb), and a *kji*-ordered, forward-looking Gaussian elimination with *ji*-backward substitution [7]. The run-times are listed in Table 4. They show that the *kji*-elimination and *ji*-backward substitution lead to substantially better processor utilization, and that they have a higher potential for compiler optimizations. However, we will not elaborate these aspects further. Instead, we focus on the analysis of the two data-parallel programs developed in Section 3.2, subsection "Gaussian Elimination." We continue with the

Table 4. Sequential Execution Times (Seconds) of Gaussian Elimination with Partial Pivoting and Backward Substitution

Optimization	Gelim *kij*	Backsb *ij*	Gelim *kji*	Backsb *ji*
-O	664.33	0.69	135.60	0.19
-O4 -Mvect -Mstreamall	583.81	0.49	55.88	0.01

Note: The large variation in the run-times shows the sensitivity of the performance with respect to compiler optimizations and loop ordering.

kji-ordered, forward-looking Gaussian elimination because of its better processor utilization, and with the *ij*-backward substitution because of its lower communication complexity for the chosen distribution (see the end of this section).

The implementation of the different strategies was hindered by several xHPF-specific problems. Unfortunately, these problems obscure the usefulness and importance of the data parallel implementation, which is our main point in discussing Gaussian elimination.

The *local pivot search*-version does not improve the performance. The xHPF compiler still broadcasts all intermediate results of the distributed array elements `ppiv(piv)` and `abspiv(piv)` in the pivot search. Because these elements are local to the processor that executes the loop, we did not expect that the compiler would insert this communication inside the loop, since the resulting values after the loop execution are important for the rest of the algorithm and the other processors. We can avoid the communication inside a loop with the xHPF parallelizer by using a forced optimization that inhibits this communication. However, this leads to a segmentation violation in one of the APR run-time library routines if the program is executed on more than one processor. With several other "forced optimizations" and algorithmic changes we could get the program to run, but we never obtained reasonable timings for the *local pivot search*-version. We would like to emphasize that such an implementation, with many forced optimizations, is highly undesirable.

The *all local*-version does not lead to the desired efficiency immediately with the xHPF compiler. If array `pcol` is declared as an automatic array inside the subroutine `gelim`, the copying of the pivot vector (Loop 400 in the Appendix, "Data-Parallel Linear Equation Solver, All Local Variant") is implemented by broadcasting each vector element separately instead of broadcasting the whole vector once after the copy. Furthermore, a subroutine call for computing the local indices for the replicated array `pcol` is inserted inside the loop. Hence, the xHPF compiler not only creates unnecessary communication, but also introduces unnecessary work. This obviously leads to very poor performance, which is completely unnecessary (the PGI compiler does not have these problems). The execution times are shown in Table 5 in the "Automatic Array" column.

To improve the performance we used the following tactic. If array `pcol` is allocated in the calling routine and passed as a dummy array to `gelim`, the pivot column is broadcast as a whole, and the index calculation is moved out of the loop. This also shows that there is no good reason for the poor implementation by the xHPF compiler when an automatic array is used. The implementation with the dummy-replicated array is given in the Appendix (see "Data-Parallel Linear Equation Solver, All Local Variant" and the (much better) timings are given in Table 5 in the "Dummy Array" column. This type of compiler anomalies is obviously highly undesirable.

After Gaussian elimination we compute the solution by backward substitution. For the given distribution of the matrix, there are two straightforward implementations: A row-oriented (*ij*-version) and a column-oriented (*ji*-version) backward substitution with the solution vector `x` aligned with the columns of the matrix. The column-oriented version generates more communication than the row-oriented version. The column-oriented version requires a broadcast of the array element `x(col)` and a one-to-all scatter of `a(1:row-1,col)` in each step, whereas the row-oriented version needs only a reduction add over the local products `a(row,col)*x(col)`, `col=row+1, . . . , n`, which costs about the same

Table 5. Execution Times (Seconds) of Data-Parallel Style Gaussian Elimination with Partial Pivoting and Backward Substitution, *all local* Version

No. of Procs	Automatic Array		Dummy Array	
	Elim	Back	Elim	Back
1	326.06	1.06	136.81	1.07
2	703.54	1.17	72.02	1.15
4	1297.54	1.34	39.81	1.36
8	3009.48	1.75	24.06	1.78
16	7203.39	2.49	17.50	2.52
25	—	—	16.20	3.31
32	—	—	16.31	3.69
50	23096.44	10.51	18.58	5.04
64	—	—	20.31	5.99
80	—	—	23.28	7.13
96	—	—	26.59	8.18

Table 6. Execution Times (Seconds) for the Finite-Volume Discretization and 25 Iterations in Total of the Nested Iterative Solver for Three Problem Sizes: The Sequential Program and the HPF Program for Several Numbers of Processors

	$4 * 10^4$		$1 * 10^5$		$2.5 * 10^5$	
No. of Procs	discr.	it.solver	discr.	it.solver	discr.	it.solver
1 (seq)	0.38	5.20	1.00	13.24	2.38	42.38
1 (hpf)	0.74	15.23	1.84	37.19	4.58	112.78
2	0.48	8.24	1.11	19.23	2.71	81.81
4	0.23	4.69	0.54	10.30	1.35	66.65
8	0.19	3.12	0.37	6.61	0.65	14.47
16	0.14	2.61	0.22	4.06	0.42	8.36
32	0.09	2.69	0.17	3.49	0.20	5.36
64	0.14	3.69	0.18	4.26	0.21	5.32
96	0.18	4.78	0.18	5.16	0.22	5.83

(with respect to communication) as the broadcast of a scalar. Therefore, we have chosen the row-oriented version for our results.

Flow-Simulation

The execution times for the discretization and a fixed number of steps of the iterative solver in the final HPF program are given in Table 6 for three problem sizes and various numbers of processors. For one processor, Table 6 gives the run-time for the original sequential code (seq) and for the HPF code (hpf) executed on one processor. For the first two problem sizes the program fits in the memory of a single processor, for the largest problem size the program does not fit in the memory of a single processor (so some paging is done) but fits in memory for two processors or more. So, considering the size of the machine, 96 processors, this is still a small problem.

It is clear from a comparison between the sequential execution times of the original program and the HPF program that the latter induces a substantial amount of overhead. Since no special measures are taken in the solver to improve the communication effects of a large number of inner products, which need global communication, the speedup deteriorates rapidly for smaller problems [9], [16]. However, for the largest problem the speedup on 32 processors is approximately eight. Moreover, if we look at the three problem sizes for a specific number of processors (say 64), we see that the execution time increases by 25% to 35%, whereas the problem size increases by a factor of 2.5. So for very large problems the overhead induced by the compiler becomes less noticeable, and the (relative) performance improves.

3.4 Review of Experimental Analysis

Dense Matrix-Matrix Multiplication

The performance study in Section 3.3, subsection "Dense Matrix-Matrix Multiplication" leads to the identification of two efficiency-related issues of the parallel program generated by HPF:

1. Loop parallelization match: Consistency of the data distribution with the loop order and loop parallelization. Depending on the loop nesting order and the given data distribution, an HPF source-to-source compiler inserts run-time checks, index transformations, and communication primitives. The location of this code determines the number of messages generated, and hence affects the run-time efficiency. Given the data distribution, the appropriate order of the three nested loops in the sequential program is essential to ensure minimal communication overhead, and vice versa. We call this aspect the loop parallelization match to characterize the match of the data distribution with the loop order and loop parallelization.
2. Optimization gap: Loss of optimization potential of the parallel program generated by an HPF source-to-source compiler. Efficiency aspects related to the processor architecture, in particular, register allocation, pipelining, and caching, are coupled with the loop order. Complex RISC processors such as the i860 of the Paragon rely heavily on machine-specific optimizing compilers for generating efficient sequential code [17]. In our case, the quality of the sequential code for the individual processors depends on the

ability of the native compiler to optimize the program generated by the HPF compiler. We call the interface problems between the two compilers the optimization gap.

Although the optimization gap is not HPF specific, there are two reasons to mention this aspect explicitly. First, there is a large difference in performance between the HPF programs and the sequential program on one processor. Second, there is also a difference in performance between the HPF programs with different loop orders on multiple processors. Furthermore, the loop parallelization match and optimization gap are closely coupled, which complicates the analysis. The optimization gap is illustrated in Table 2, the loop parallelization match is illustrated in Table 3, and an overview of the joint effects can be obtained from Figure 6.

An important problem with the analysis of the timing results is the lack of knowledge of vendor- or compiler-specific subroutine and function calls to runtime libraries inserted by HPF compilers. Often the only feasible way to analyze the peculiarities will be performance monitoring or empirical methods. For the simple matrix-matrix multiplication code, it is possible, although time consuming, to obtain an empirical performance analysis by gathering data of the complete set of loop permutations. These experiments show that the loop parallelization match and the optimization gap strongly influence the parallel efficiency. Moreover, Figure 6 indicates that the optimal loop order depends on the number of processors. Hence, there is no variant that is optimal for all numbers of processors. Experimenting and profiling may be the only way to find the best variant for a particular number of processors, since it seems difficult to predict.

Gaussian Elimination

The programming experience with Gaussian elimination illustrates the kind of considerations that are necessary in general to obtain an efficient parallel implementation. One should not confuse these considerations with the xHPF-specific implementation problems.

The programmer has to present a data-parallel program to the HPF compiler. In general, this requires the restructuring of the original F77 program, adding or modifying data structures and making (some) algorithmic changes. This should and will, in general, guarantee good performance irrespective of the compiler. It is highly undesirable that certain constructs lead to good performance on one compiler and poor performance on another (performance portability). Also it should be clear which constructs work and why, and the difference between good and bad performance should not depend on trivial changes to the code, which is sometimes the case with the xHPF compiler (as discussed previously).

The xHPF compiler, however, even with a data-parallel programming style, may not generate an efficient parallel program. In such a case, the programmer has to study carefully the parallelization report and the xHPF-generated program, and either provide APR-specific directives that force the compiler to insert or omit communication code or make (small) changes to the (HPF) source code that prevent erroneous or poor implementations. Using forced optimizations should be viewed as supporting the xHPF compiler to overcome the limitations in its optimization capabilities, especially at this (early) stage of parallelizing compiler technology. However, the use of forced optimizations is dangerous; it may change program semantics, and it is not portable to other HPF compilers.

Flow-Simulation

Before we discuss our conclusions concerning the four problem types discussed in Section 3.2, subsection "Flow-Simulation," we consider the type of changes in declarations that are important when converting F77 to HPF, including some issues that do not arise in a F77 program at all. HPF incorporates F90 [18], and hence it supports several dynamic allocation features. This removes the need for many practices in F77 that make automatic parallelization difficult, apart from the problems that they pose for programming in general.

It will no longer be necessary to declare arrays larger than needed in the actual invocation of a program or subroutine. If the required size is not known in advance, the array can be made allocatable and allocated once the required size is known. If the required size is passed through the argument list of a subroutine, the array can be allocated inside the subroutine with this size (automatic array). Therefore, dummy workspace arrays are no longer necessary. Furthermore, dummy arguments can have undefined shape, which means that they will take their shape from the actual argument (assumed shape array). Finally, it is possible to define (declare) the distribution/alignment attributes of an array before the array has actually been allocated.

Also common blocks, a major source of problems in HPF (but often in F77 also), are no longer necessary. Global variables can be created and managed (for access) more cleanly using MODULEs and the USE statement [18]. Having a single subroutine perform

the same (or similar) function on a range of different objects, e.g., matrices in different storage formats, can be achieved either by using optional keywords or by using generic interfaces (overloading) [18]. Stricter definitions/declarations of data and specifying exact sizes will make compiler optimizations easier, communications cheaper, and it will remove load-balancing problems.

An important point, especially with the F90 (dynamic) allocation features, is where arrays should be declared. From a software engineering point of view, data should be declared at the (highest) level where it is meaningful and where it makes sense to make this data visible and allow manipulation. Also from a parallelization point of view this is generally the best. However, for the xHPF compiler, sometimes large arrays that are often referenced together should preferably be declared together, because this makes their mutual alignment/distribution easier. This may mean that an array is declared at a higher level than necessary (meaningful), because this will typically reduce directive-based implicit alignment problems.

Directive-based implicit alignment problems (which only occur with the xHPF compiler) can often be removed by writing out small (often used) subroutines explicitly. This is not necessarily as bad as it sounds, because HPF (F90) is a much more powerful language than F77, and offers more flexibility and expressiveness in addressing parts of arrays. For example, most level 1 BLAS can be written in a single statement, and with the triplet notation in array syntax all the complicated increment (stride) issues can be avoided.

Another important issue is that the potential optimization to move (re)distributions of arrays on entry to and exit from a subroutine out of the subroutine is not always useful. The redistribution may be useful only inside that subroutine, and, if the (re)distribution takes place on entry and the original distribution is restored on exit, no implicit alignments outside the subroutine can arise. We note that the APR xHPF compiler has no choice but to move distributions out of subroutines (i.e., force the appropriate distribution to be performed where the array is declared), because it does not support restoring the original distribution on exit from a subroutine. This is a severe drawback and the main reason for the directive-based implicit alignment problems. Moreover, this implementation does not conform to the official HPF Subset definition.

The main way to prevent loop-based implicit alignment problems is cloning where a subroutine may use distributed arrays as actual arguments with a different shape, size, or distribution in a loop. This means that it may be better to have two routines that perform the same or similar actions, but place different requirements on their arguments. Of course, it would be much better if the compiler handled such problems transparently. Furthermore, using arrays with different distributions in a single loop should be avoided if possible. Writing out small, often used, subroutines may also relieve problems. Both explicit cloning and inlining should be considered as temporary work-arounds to cope with the problems of currently available compilers. Eventually, compilers should take care of these problems transparently. Another important issue is to reduce the complexity of loop structures and index expressions. Finally, one should be careful with nested loop structures.

The aliasing problems can often be avoided using the derived types and POINTER features of F90/HPF (not in the HPF Subset). Careful declaration will avoid several problems. Finally, using array syntax (segments and triplet notation) may avoid aliasing problems. Better compiler analysis, on the other hand, will also improve the situation.

Where the analysis capabilities of the compiler are insufficient, interprocedural analysis should be abandoned or limited, because it causes more problems than it solves. Once more, it is important to note that in several cases, the xHPF compiler has to rely on interprocedural analysis, because its handling of distributed data over subroutine boundaries is nonstandard and can create problems if distribution of actual arguments is not moved to the subroutine where the arrays are declared. This approach has the additional drawback that if distribution of the actual arguments is not possible then the desired distribution must be ignored. In cases where the interprocedural analysis fails, it is much more useful to use inquiry routines to determine the run-time configuration and data layout and to use this information to implement the desired strategy explicitly. (xHPF does not provide the HPF inquiry routines; in the new release only one APR-specific and limited distribution inquiry routine is provided.)

4 DISCUSSION

In this section we discuss some general problems related to the performance portability of the HPF programs and summarize possible solutions. Our recommendations are based on the work with APR's xHPF compiler, although we try to generalize them by considering their background. Furthermore, we encountered several other problems typical of the APR compiler, which we will not discuss.

4.1 Performance Portability

Currently, only an HPF language specification has been defined, which (of course) does not include a specification of the optimization capabilities of an HPF compiler. This allows a wide range of possible HPF compiler implementations, which poses a threat to the goal of performance portability. Preliminary testing of our programs using the PGI* compiler confirmed our suspicion that for the same program different compilers generate codes delivering very different performance; i.e., the PGI compiler sometimes creates poor code where the xHPF compiler creates efficient code and vice versa. The HPF language specification allows the programmer to make few assumptions about the generated code. At the main program level, all directives (particularly distributions and alignments) are so-called prescriptive directives, i.e., they indicate a desired feature but they may be ignored. Strictly speaking, the programmer has no control over the actual distribution, although in practice the situation will not be so bad. Nevertheless, any distributions that the compiler will use or "grant" the programmer and any optimizations it will make depend on the analysis capabilities of the compiler.

As an example of the range of possible compiler implementations, consider the following case. In a subroutine the programmer asserts the alignment of two dummy arguments (a descriptive alignment). It is therefore the programmer's responsibility that this alignment holds upon each subroutine entry. However, in the main program, the programmer cannot insist on any particular distribution, because only prescriptive directives can be used, which may be ignored by the compiler. Recognizing this problem, the authors of the HPF Language Specification [2] provide the following statement:

> All this† is under the assumption that the language processor has observed all other directives. While a conforming HPF language processor is not required to obey mapping directives, it should handle descriptive directives with the understanding that their implied assertions are relative to this assumption.

We can distinguish two entirely different ways of handling this problem at the compiler level. First, when the compiler does not observe even a single directive or encounters even a single analysis problem, it simply ignores all descriptive directives. Second, the compiler performs global interprocedural analysis and propagates all requirements on dummy arguments to the actual arguments in the calling routine(s). In this way the compiler can check whether decisions it has made elsewhere in the program have an influence on this particular descriptive directive. The directive may even influence distribution decisions taken at higher levels in the program. Obviously, there are many possibilities between the two extremes. These possibilities are determined by the quality of the analysis and the type of analysis being performed, and by the optimizations that the compiler supports.

4.2 Hints for Programming with HPF

In order to support novice users of HPF, we provide a summary of those points of the parallelization process that appear most important to us.

One should always start with a sound analysis of the program with respect to parallelization, the algorithms involved, and the purpose of the program.‡ From this analysis, a parallelization strategy should be developed that involves the distribution of both data and work. The distribution of data is done through declarations and directives (`distribute` and `align`). The distribution of work is done implicitly by using a data-parallel programming style, by using statements and constructs that indicate independence of loop iterations (`forall` and `independent`, and array syntax), and by using routines from the HPF_LIBRARY module and HPF intrinsics (not in the HPF Subset). In order to assist the compiler in generating an efficient parallel program, one should reduce as much as possible the complexity of nested loop structures and index expressions. Furthermore, declaring arrays only where they are needed and using the dynamic allocation features generally improve the generated parallel code. However, for the xHPF compiler, one should be aware that sometimes it is better to declare large, distributed arrays at a higher level in the program to facilitate the alignment with other arrays.

Taking care of the loop parallelization match means programming loop nestings and array distributions such that communication can be moved out of the loops as much as possible. Decreasing the number of subroutine calls to the vendor-supplied run-time library for communication and run-time checks inside the loops will also increase the optimization opportuni-

* Trademark of Portland Group Inc.

† It is the programmer's responsibility that the asserted statement holds.

‡ We mention this obvious fact, because we feel that recently HPF has been oversold to be very easy to use, and to involve nothing more than inserting a few directives at the start of a program or subroutine. This, however, is not our experience.

ties for the native compiler, which may reduce the optimization gap. The optimization gap may be further reduced by adjusting the loop order in the (original) F77 or HPF program such as to improve locality, vector length, or other features in the program generated by the HPF compiler. However, one should be aware that such changes may improve the performance on one parallel machine but may reduce it on another parallel machine. In general, such optimizations require a thorough understanding of the HPF compiler itself, or at least extensive experience with its black-box behavior. In many cases, loop parallelization match and optimization gap requirements will conflict. One possibility to resolve this is to change the algorithm and/or the data structures. In numerical simulations, different algorithms often can be used to obtain the same result, and the choice for the current algorithm may have been based on assumptions that do not hold in an HPF setting.

Another important consideration is that on a machine with a small number of powerful processors, optimizations by the native compiler might be more effective than minimizing the cost of communication, whereas on a machine with many relatively slow processors the opposite might be true. Changing the algorithms and data structures or replacing one algorithm by another is useful when simple "tricks" do not work.

With the xHPF compiler, it is important to prevent problems with directive-based implicit alignments. Directives for dummy (array) arguments in subroutines should be used carefully, and careful handling of the actual arguments in the subroutine calls is recommended to avoid unnecessary redistribution. Often alignments can be made at a higher level to prevent some of the implicit alignment problems.

To resolve problems with loop-based implicit alignments one could consider the inlining of small and frequently called subroutines. Another way to support the compiler is to provide different versions of certain subroutines for array arguments with different distributions. These suggestions should be considered as temporary solutions to cope with current compiler inadequacies. Eventually compilers should resolve (potential) problems of this type transparently. In general, one should prevent loops referencing arrays that cannot be aligned.

Alias problems can sometimes be prevented by using automatic arrays or array segments; one should not pass workspace arrays. Another possibility is to use pointers, although this only helps in simple cases. With the xHPF compiler, one can use forced optimizations if changing the algorithms and data structures does not help.

To avoid interprocedural propagation problems with the xHPF compiler, one should try to remove the problem that is the source of the propagation. To remove remaining problems one should again consider changes to algorithms and/or data structures. Finally, one could use forced optimizations.

With the xHPF compiler one could, at the end of the parallel program development, consider the use of forced optimizations to improve the efficiency of the parallel code by removing unnecessary communication and run-time checks.

5 CONCLUSIONS

Our case study revealed several problems with the conversion of F77 programs to HPF programs: the loop parallelization match, the optimization gap, program changes to support the compiler with program sections whose parallelization is not clear from the data distribution, directive-based implicit alignment (xHPF specific), loop-based implicit alignment, alias problems, and propagation problems introduced by the interprocedural analysis (xHPF specific). From our experience with the PGI compiler and from discussions with several other people involved in HPF [T. Brandes, Personal Communication; L. Meadows, Personal Communication; R. Rühl, Personal Communication], it appears that, except for the directive-based implicit alignment and the propagation problems introduced by the interprocedural analysis, the problems that we indicated are of general importance. Here, "general importance" does not indicate that these problems are inherent to the HPF language, but indicates that they arise in several currently available HPF compilers, and are not easy to resolve within the compiler. For most of these problems, we have indicated possible solutions at the program level (although they do not always work with the xHPF compiler). The solution of the identified problems at the compiler level would enhance the performance of HPF programs and substantially reduce the effort of the programmer. We obtained reasonable performance for the programs in our case study, but only with several changes in the programs and algorithms.

Concerning the APR xHPF compiler we observed the following. In general, an HPF Subset program must be (re)written in a data-parallel programming style. Furthermore, a rather detailed understanding of the compiler is necessary to obtain good results. This requires patience and time in order to gather sufficient experience. The compiler is sometimes overconservative (which probably indicates insufficient analysis capabilities), so that considerable time must be devoted to removing unnecessary communication

calls and run-time checks when parallelizing a particular program.

REFERENCES

[1] APR, *FORGE HPF Parallelizer xHPF User's Guide 1.0.* Applied Parallel Research, 550 Main Street, Suite I, Placerville, CA 95667, 1993.

[2] "High Performance Fortran language specification," *Sci. Prog.*, vol. 2, 1993.

[3] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele, Jr., and M. E. Zosel, *The High Performance Fortran Handbook.* Cambridge, MA: MIT Press, 1994.

[4] J. Merlin and A. Hey, "An introduction to High Performance Fortran," *Sci. Prog.*, vol. 4, pp. 59–85, Summer 1995.

[5] P. Arbenz, "First experience with the Intel Paragon, in 16th Speedup Workshop on Vector and Parallel Computing," *Speedup*, vol. 8, pp. 53–58, Dec. 1994.

[6] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for Fortran usage," *ACM Trans. Math. Software*, vol. 5, pp. 308–323, Sept. 1979.

[7] T. L. Freeman and C. Phillips, *Parallel Numerical Algorithms.* Englewood Cliffs, NJ: Prentice Hall, 1992.

[8] E. De Sturler, "Nested Krylov methods based on GCR," Faculty of Technical Mathematics and Informatics, Delft University of Technology, Delft, The Netherlands, Tech. Rep. 93-50, 1993. (Accepted for publication in the *J. Comp. Appl. Math.*)

[9] E. De Sturler and H. A. van der Vorst, "Reducing the effect of global communication in GMRES(m) and CG on parallel distributed memory computers," *Appl. Numerical Math.*, vol. 18 (IMACS), pp. 441–459, 1995.

[10] P. J. Hatcher and M. J. Quinn, *Data-Parallel Programming on MIMD Computers.* Cambridge, MA: MIT Press, 1991.

[11] W. D. Hillis and G. L. Steele, Jr., "Data parallel algorithms," *Commun. ACM*, vol. 29, pp. 1170–1183, Dec. 1986.

[12] H. Zima and H. and B. Chapman, *Supercompilers for Parallel and Vector Computers.* New York: ACM Press Frontier Series (Addison-Wesley, Wokingham, 1991).

[13] R. Rühl, "A parallelizing compiler for distributed memory parallel processors," Ph.D. thesis, ETH Zürich, 1992.

[14] R. Ponnusamy, J. Saltz, and A. Choudhary, "Runtime-compilation techniques for data partitioning and communication schedule reuse," University of Maryland, Tech. Rep. UMIACS-TR-93-32.1, Oct. 1993.

[15] T. Brandes and F. Zimmermann, "*Data Parallel Programming on IBM's Scalable Parallel Systems with the ADAPTOR Tool*, 1994. (URL http://www.gmd.de/SCAI/lab/adaptor/documents.html).

[16] E. De Sturler and H. A. van der Vorst, "Communication cost reduction for Krylov methods on parallel computers," in *High-Performance Computing and Networking*, Lecture Notes in Computer Science 797, W. Gentzsch and U. Harms, Eds. Berlin: Springer-Verlag, 1994, pp. 190–195.

[17] D. H. Bailey, "RISC microprocessors and scientific computing," in *Supercomputing '93*, 1993, p. 645.

[18] M. Metcalf and J. Reid, *Fortran 90 Explained.* Oxford: Oxford University Press, 1994.

APPENDIX

Linear Equation Solver

This is the sequential implementation with HPF directives for data distribution and with APR-specific forced optimizations (see Table 1 for timing results).

```
 1 c
 2 c     Gaussian elimination with partial pivoting
 3 c
 4       subroutine gelim(a,lda,b,n,tol,perm,errflg)
 5 c
 6 c     *** in/out variables ***
 7 c
 8       double precision a(lda,n), b(n), tol
 9       integer          lda, n, perm(n), errflg
10 c
11 c     *** local variables ***
12 c
13       integer          row, col, piv, ppiv, itmp
14       double precision abspiv, dtmp
15 c
```

```
16 c     *** distribution and alignment ***
17 c
18 chpf$ distribute a(*,cyclic)
19 chpf$ distribute b(*)
20 chpf$ distribute perm(*)
21 c
22       errflg = 0
23       do 200 piv =1, n-1
24 c
25 c        *** pivot search ***
26 c
27          ppiv = piv
28          abspiv = abs(apiv,piv))
29 capr$    do par on a<:,piv>
30 capr$    ignore sync com
31          do 110 row = piv+1, lda
32             if (abs(a(row,piv)) .gt. abspiv) then
33                abspiv = abs(a(row,piv))
34                ppiv   = row
35            endif
36  110     continue
37 c
38 c        *** permutation for perm ***
39 c
40          itmp       = perm(piv)
41          perm(piv)  = perm(ppiv)
42          perm(ppiv) = itmp
43 c
44 c        *** check for singularity ***
45 c
46          if (abspiv .lt. tol) then
47             errflg = ppiv
48             write (*,*) 'tolerance exceeded: ', abspiv
49             return
50          else if (ppiv .ne. piv) then
51 c
52 c           *** permutation for a ***
53 c
54 capr$       do par on a<:,col>
55             do 120 col = 1, n
56                dtmp        = a(piv,col)
57                a(piv,col)  = a(ppiv, col)
58                a(ppiv,col) = dtmp
59  120        continue
60 c
61 c           *** permutation for b ***
62 c
63             dtmp   = b(piv)
64             b(piv)  = b(ppiv)
65             b(ppiv) = dtmp
66          endif
67 c
68 c        *** elimination ***
69 c
```

```
 70 capr$    do par on a<:,piv>
 71 capr$    ignore all com
 72          do 125 row = piv+1, n
 73             a(row,piv) = a(row,piv) / a(piv,piv)
 74  125     continue
 75 capr$    do par on a<:, piv+1~1>
 76 capr$    ignore all com on a<1+piv~1,piv>
 77          do 140 col = piv+1, n
 78             do 130 row = piv+1, n
 79                a(row,col) = a(row, col) - a(row,piv) * a(piv,col)
 80  130        continue
 81             b(col) = b(col) - a(col,piv) * b(piv)
 82  140     continue
 83 c
 84  200  continue
 85 c
 86 c     *** final check for singularity ***
 87 c
 88       if (abs(a(n,n)) .lt. tol) then
 89          errflg = n
 90       endif
 91       end
 92 c
 93 c     Backward substitution
 94 c
 95       subroutine backsb(a,lda,b,x,n)
 96 c
 97 c     *** in/out variables ***
 98 c
 99       double precision a(lda,n), b(n), x(n)
100       integer          lda, n
101 c
102 c     *** local variables ***
103 c
104       integer          row, col
105 c
106 c     *** distribution and alignment ***
107 c
108 chpf$ distribute a*(*,cyclic)
109 chpf$ distribute b*(*)
110 chpf$ align x(col) with *a(x, col)
111 c
112       do 330 row = 1, n
113          x(row) = b(row)
114  330  continue
115       do 350 col = n, 1, -1
116          x(col) = x(col) / a(col,col)
117          do 340 row = 1, col-1
118             x(row) = x(row) - a(row,col) * x(col)
119  340     continue
120  350  continue
121 c
122       return
```

```
123       end
124
```

Data-Parallel Linear Equation Solver, All Local Variant

Pivot, search, row exchange, and elimination are independent for all (distributed) columns: (see Table 5 for timing results).

```
 1       subroutine gelim(a,lda,b,n,tol,per,pcol,errflg)
 2 c
 3 c     *** in/out variables ***
 4 c
 5       double precision a(lda,n), b(n), tol
 6       integer          lda, n, perm(n), pcol(n), errflg
 7 c
 8 c     For programming reasons pcol should have been
 9 c     an automatic array; however, in that case the xHPF compiler
10 c     would compute indexes for pcol inside the copy-loop.
11 c
12 c     *** local variables ***
13 c
14       integer          row, col, piv, ppiv, itmp, npiv
15       double precision abspiv, dtmp
16 c
17 c     *** intrinsic functions ***
18 c
19       intrinsic        abs
20       double precision abs
21 c
22 c     *** distribution and alignment ***
23 c
24 chpf$ distribute a(*,cyclic)
25 chpf$ distribute b(*)
26 chpf$ distribute pcol(*)
27 chpf$ distribute perm(*)
28 c
29       do 200 piv = 1, n-1
30 c
31 c        *** copy pivot column into pcol (replicated) ***
32 c
33          do 400 row = 1, lda
34             pcol(row) = a(row,piv)
35  400     continue
36 c
37 c         *** pivot search ***
38 c
39          ppiv = piv
40          abspiv = abs(pcol(piv))
41          do 110 row = piv+1, lda
42             if (abs(pcol(row)) .gt. abspiv) then
43                abspiv = abs (pcol(row))
44                ppiv   = row
45              endif
46  110      continue
```

```
 47 c
 48 c         *** check for singularity ***
 49 c
 50           if (abspiv .lt. tol) then
 51             errflg = ppiv
 52             write (*,*) 'tolerance exceeded: ', abspiv
 53             return
 54           else if (ppiv .ne. piv) then
 55 c
 56 c            *** permutation in a ***
 57 c
 58             do 120 col = 1, n
 59                dtmp        = a(piv,col)
 60                a(piv,col)  = a(ppiv,col)
 61                a(ppiv,col) = dtmp
 62  120        continue
 63 c
 64 c            *** permutation in b ***
 65 c
 66             dtmp    = b(piv)
 67             b(piv)  = b(ppiv)
 68             b(ppiv) = dtmp
 69 c
 70             *** permutation in perm ***
 71 c
 72             itmp       = perm(piv)
 73             perm(piv)  = perm(ipiv)
 74             perm(ipiv) = itmp
 75             *** permutation in pcol ***
 76             dtmp       = pcol(piv)
 77             pcol(piv)  = pcol(ipiv)
 78             pcol(ipiv) = dtmp
 79           end if
 80 c
 81           *** elimination ***
 82 c
 83           do 125 row = piv+1, lda
 84              pcol(row) = pcol(row) / pcol(piv)
 85  125      continue
 86           do 140 col = piv+1, lda
 87              do 130 row = piv+1, n
 88                 a(row,col) = a(row,col) - pcol(row) * a(piv,col)
 89  130         continue
 90  140      continue
 91           do 150 row = piv+1, lda
 92              b(row) = b(row) - pcol(row) * b(piv)
 93  150      continue
 94 c
 95  200  continue
 96 c
 97 c      *** final check for singularity ***
 98 c
 99        if (abs(a(n,n)) .lt. tol) then
100           errflg = n
```

```
101       endif
102 c
103       return
104       end
```

Propagation

The interprocedural analysis propagates parallelization problems.

```
 1       subroutine gcro(...)
 2 c
 3       double precision r(n), uu(n,m+1)
 4       double precision x(n), cc(n,m+1)
 5       double precision ri(n), vv(n,m+1)
 6       double precision hh(m+1,m)
 7 c
 8 chpf$ distribute r(block)
 9 chpf$ align uu(i,*) with r(i)
10 chpf$ align cc(i,*) with r(i)
11 chpf$ align vv(i,*) with r(i)
12 c
           .
           .
           .
13       call gmreso(n,m,k,r,uu(1,k+1),ri,...)
14       k = k+1
15       do i = 1, n
16          cc(i,k) = r(i) - ri(i)
17       end do
18       cn(ksize) = sqrt(ddot(n,cc(1,ksize),1,ksize),1))
19       call daxpy(n,-d(ksize),cc(1,ksize),1,r,1)
20       call daxpy(n,d(ksize),uu(1,ksize),1,x,1)
           .
           .
           .
21       end
22 c
23       subroutine gmreso(n,m,k,b,x,r,...)
24 c
25       double precision vv(n,m+1)
           .
           .
           .
26 c
27 chpf$ inherit b
28 chpf$ align vv(i,*) with b(i)
           .
           .
           .
29 c
30 c      ***** Arnoldi *****
31       do while ((j.le.m) .and. (.not.conv))
32          call matvec(n,vv(1,j),vv(1,j+1),...)
33 c        ***** orthogonalize on cc_1 .. cc_k *****
34          do i = 1, k
35             bb(i,j) = ddot(n,cc(i,i)1,vv(1,j+1),1)/(cn(i)**2.d0)
36             call daxpy(n,-bb(i,j),cc(1,i),1,vv(1,j+1),1)
37          end do
```

```
38 c        ***** orthogonalize on vv_1 .. vv_j *****
39          do i = 1, j
40             hh(i,j) = ddot(n,vv(1,i),1,vv(1,j+1),1)
41             call daxpy(n,-hh(i,j),vv(1,i),1,vv(1,j+1),1)
42          end do
              .
              .
              .
43       end do
              .
              .
              .
44 c     ***** compute inner solution *****
45       do i = 1, j-1
46          call daxpy(n,rsh(i),vv(1,i),1,x,1)
47       end do
              .
              .
              .
48 c     ***** compute inner residual *****
49       do i = 1, j
50          call daxpy(n,-xl(i),vv(1,i),1,r,1)
51       end do
52       ***** oblique orthogonalization of new u_k+1 *****
53       do i = 1, k
54          call daxpy(n,-x0(i),uu(1,i),1,x,1)
55       end do
              .
              .
              .
56       end
```